Maintaining Sanity in a Multilanguage World

Val C. Kartchner
309 SMXG/MXDDA

The Ada 2005 standard will help many users. But the reality of working in a frozen, legacy development environment needs to be addressed. Development in a mixed version (Ada 83 and Ada 95) and mixed language (C and C++) environment involves dealing with many issues. This article addresses the issues that we encountered when developing applications for the Air Force Mission Planning System. These issues fit into three main categories: dealing with Ada strings, using inter-language interfacing, and using different Ada compilers (83 and 95) but maintaining one code base. This article discusses several of the technical issues involved in interfacing Ada, C, and C++ from both a syntactical and run-time perspective.

The Air Force uses three different mission planning systems (Mission Planning System [MPS], Portable Flight Planning Software [PFPS] and Joint Mission Planning System [JMPS]) to plan routes and missions for training and actual war fighting. The Overlay Import/Export Tool (OIET) and MPS Common Route Definition Import/Export Tool (MCIET) were developed to import to the MPS data from PFPS and JMPS, or export data from the MPS for use on PFPS and JMPS. OIET imports/exports machine and user generated graphics needed for flight planning. MCIET does the same for route information. This article addresses some of the obstacles faced during the development and sustainment of these software programs in mixed development environments.

The MPS was developed by a government contractor in the early 1990s. Because mission planning is such a critical task, stability is very important, so the development and run-time systems were frozen at that time, with the exception of some critical upgrades, like the Y2K patch. However, legacy hardware and software are not supported forever. A government contract with Sun provided for the continued availability of the old versions of the operating system and compilers. But new hardware was not easily compatible with the older operating system, and the older hardware was becoming increasingly difficult to maintain. Therefore, a decision was made in 2004 to upgrade to the current hardware, operating systems, and compilers.

New hardware and software allows a great leap forward for both the developers and users. But the criticality of mission planning prevents the users from moving until their entire planning environment is moved. Each aircraft system (i.e., B-2, B-52H, F-117) has a different mission planning environment (MPE) that consists of the core software (operating system and applications) along with additional installable software modules (ISMs) and other software. This means that the OIET and MCIET need to be maintained for both the old system and the new system until all aircraft MPEs have been upgraded to the new system.

String Issues

Because many of Ada's features were originally designed to address Department of Defense (DoD) needs, Ada handles several important features (such as strings, pointers, and memory allocation/deallocation) differently from its programming peers – languages such as C, C++, and Java. This article covers several important topics that merit separate discussion, especially in the context of not only multiple-language programming, but also in the context of a mixed-mode language environment. One of the most important issues involves the way Ada treats strings as opposed to other languages.

Ada Strings

The initial MPS development environment had been frozen with an Ada 83 compiler. Strings in Ada 83 (as in most computer languages) are arrays of characters. Ada 83 was designed with a great consistency about how arrays are handled: arrays must be defined with a particular size before they can be used; when assigning one array to another, they must be the same length and contain the same type of data. This works fine for an array of generic data but not nearly as well for strings where variable-length is normal.

When strings are passed around, variable lengths are often used. Receiving a variable-length array (including strings) as a parameter to a subprogram (procedure or function) is relatively easy to do. However, returning a variable-length array (or string) as the return value of a function or an *out* parameter of a procedure is not possible using standard Ada 83 language constructs. Some sort of substitute (or trickery) must be used to accomplish the desired result. This is why the Ada 95 standard added the unbounded-string type and the allocate-and-assign construct (*pointer: = new string'(trim_ spaces (some_ string));*), so that standard solutions will be available.

But since we needed to pass variable-length strings in Ada 83, we developed

Figure 1: *Core of VString Package*

```
package VString is
  type VString is limited private;
  subtype C_String_Ptr is CString_Interfaces.C_String_Ptr;
  …
private
  type String_Access is access String;
  type VString is
    record
      cur_length : Natural := 0;
      str_access : String_Access := Null;
    end record;
end;
```

the VString (variable string) package. A VString variable is basically a string-access (string pointer) with the current-length-used, but the VString package hides the implementation details (see Figure 1). This package also provides an interface (supporting procedures and functions) to do what is needed to VString variables. The VString type could have been implemented as a private type, but because of the string access type inside, it is not recommended.

For instance, if VString is a private type and two variables (*Aye* and *Bee*) contain the VString values of *Alphabet* and *Spelling Bee* respectively assigning Aye to Bee (or *Bee := Aye;*) would copy the values exactly (i.e, *Bee.cur_length := Aye.cur _length;* and *Bee.str_access := Aye.str_access;*) The access to the *Spelling Bee* string is now lost. This is called a *memory leak*[1]. The string reference by both *Aye* and *Bee* is also referenced (referenced two or more times) so that if *Aye* is set to *Numerics* (using VString functions), *Bee* also holds the same string. This is not usually desirable.

Also, if *unchecked_deallocation* is instantiated to free (return to available memory) the string-access memory, and both *Aye* and *Bee* are freed, the same memory will be freed twice, creating a potentially disastrous problem in the program.

This is why VString is declared as a *limited private* type, so that assignment is not allowed between two variables of the same type. With Ada95, these problems do not exist because limited-private types may have the assignment operator overloaded and tagged-types have the equivalence of destructors (through *Ada.Finalize*).

Support subprograms are used to copy Ada strings or C strings to VStrings. VStrings may also be copied to Ada strings, with automatic truncation or extension (space filling) to the size of the destination string. A VString is also kept null-terminated[2] so that it can be passed to a C or C++ function. As needed, other support subprograms to compare, paste, slice, search, replace, output, and debug, have been added to the package. This package has proven useful and is used extensively in OIET and MCIET.

There are some good reasons for allocating more character space (in the string) than will be immediately needed. Dynamic memory allocation algorithms will always allocate memory in multiples of a *minimum allocation unit*, so taking advantage of this does not consume more actual memory than usual. Also, depending on the program and string, a string may change size several times over its lifetime and allocating a little more memory is likely to delay the need for reallocation of memory as the string grows. If the length of the string decreases, it may only be a temporary decrease, so holding on to the memory is not likely to adversely affect performance. When Ada calls for the actual string, the slice of the allocated string that is in use is returned.

Because a limited private type cannot be automatically copied (in Ada 83), we had to find another way of including variable-length strings in nodes of a generic linked list package. Instead, we used string access types directly in the nodes of the list. Another option would have been to provide a copy procedure as one of the parameters to the generic linked list package.

This VString package worked for us because we had to use Ada 83. But if you have the option, use the improved string handling capabilities added in Ada 95 or Ada 2005.

"Another key to interfacing with another language is understanding how each language will pass subprogram parameters."

More on C Strings

Ada83 has no standard way to pass or receive C strings, but the compiler that we used provided a proprietary method of doing this. The Ada95 standard has defined the *Interfaces.C.Strings* package as this interface. Passing and receiving C strings is done extensively throughout the Ada portion of our code; using the respective compiler-provided interfaces would split our code into pairs of files that would be mostly identical, thereby increasing maintenance costs. Also, using a file preprocessor would have been unwieldy in this case.

The least disruptive method to pass and receive strings was to create two different packages, one for each Ada (83 and 95) compiler. Each package would reside in a different file but internally call itself *CString_Interfaces*. Each package would present the same interface to users of the package and become an intermediary to the actual functionality provided by each compiler. Relatively small changes were then made to the code that needed to interface with C strings (including the VString package discussed above). One of these changes was to use *C_String_Ptr* wherever a pointer to (address of) a C String was needed. Any user of this package can use *C_String_Ptr* without knowing (or caring) what the actual type is. As needed, other subprograms to perform needed functions, like copying to and from C Strings, are provided in the *CString_Interfaces* package. The specification and body for each of the two implementations resides in the same directory and a *make*[3] file is used to compile the appropriate version of the package.

Interlanguage Interfacing

Since some of OIET and MCIET are written in C, strings are also often passed between Ada and C, but Ada strings are not simple. It is easier to pass C Strings using the C-String interface package provided with your compiler, as discussed in the previous section. When passing a string to a C function, unless special considerations have been made (like passing in the length also), each string will need to be null-terminated. To facilitate this, the VString package places an *ASCII.null* after the used portion of each VString. When the *cptr* function is called, it returns a *C_String_Ptr* ready to be passed into a C function. Like the *C_String_Ptr* used, it is good practice to declare a new type for each pointer type that is passed around so that a user does not accidentally pass a pointer of one data type to a function expecting another.

Also, do not assume that an *integer* in Ada will correspond to an *int* in C. Depending on the compiler and compiler options, it may or may not. But finding out which types of addresses, integers, and floating-point values correspond between Ada and C is a simple process of consulting the respective compiler manuals.

Another key to interfacing with another language is understanding how each language will pass subprogram parameters. FORTRAN always passes parameters *by reference*. That is, if *1* is passed to a subprogram, the value of *1* is placed in memory and the address of the memory location is passed to the receiving subprogram (in a processor register or on the processor stack). C passes parameters *by value* or *by address*. Passing by value means that if a value of *1* is passed as a

parameter; a *1* is passed to the receiving subprogram. Passing by address means that the same thing happens as passing *by reference*. C++ may pass *by value*, *by reference,* or *by address*.

The difference between *by address* and *by reference* is determined by how the receiving program treats the address received. If the parameter is received *by reference*, then the receiving subprogram knows to fetch the value from the address specified. That is, the address of the variable is implicitly *dereferenced*. If the subprogram receives *by address*, it must specify that it is fetching the value from the address. That is, the address of the variable is explicitly *dereferenced*. This applies to Ada when an access type is passed into a subprogram and .all is used to get the value stored at that access location.

It is important to understand the different ways that parameters may be passed to discuss how to interface between languages. In both of the Ada compilers used for this project (Ada 83 and Ada 95), a parameter specified as *in* is passed *by value*, and a parameter specified as *out* or *in out* is passed and received *by reference*. Simple types (integer, floating-point numbers, addresses, etc.) are returned from Ada and C functions *by value*. Ada may return complex types (strings, records, arrays, etc.), but we did not experiment with returning such types between languages.

Concerning subprogram parameters, Ada is more restrictive than C. Ada procedures allow parameters to be *in*, *out*, or *in out*, but allow no value to be returned. Ada functions have return values, but parameters can only be *in*, which is also the default if not specified. It is best to use the more restrictive Ada rules. In C or C++, the equivalent to an Ada procedure would be a *void* function. If a C function needs to have both *out* parameters and a return value (such as a system function), then a wrapper function can be used.

For instance, to find the status of a file, the system function *int stat(const char *path, struct stat *buf);* is used. Success or failure status is returned from the function as an integer value, and the status of the file itself is returned in the buffer. A C wrapper function could look like this: *void wrap_stat(int* result, const char *path, struct stat *buf) { *result = stat(path, buf); }*. The Ada declaration to call this C function would look like *procedure wrap_stat(result : out integer; path: in C_String_Ptr; buf : out stat_record_type);* with the supporting declarations of the *stat_record* and C function.

Ada and C++ Issues

One problem between Ada and C++ surfaces during runtime. Both Ada and C++ have variables, records, and objects that must be initialized when the program starts to run, before the first subprogram (*main* in the case of C++) is entered. In order to do so, each language wants to control program start-up, but only one can. An alternative considered was to let one of the languages start the program then call the other language's *initializer* (subprogram to initialize data). This varies from compiler to compiler and even operating system to operating system. These routines were not found in the manuals for our compilers, so another plan had to be devised.

The program was broken into two pieces: one program initialized by Ada and one program initialized by C++. The Ada portion runs the Graphical User Interface (GUI) and calls the C++ portion much like a subroutine would be called. This is implemented by using the Unix system functions *fork* (to start another child process), *execl* (to execute a new program inside of a process), and *wait* (to wait for a child process to complete). The exit status of the subprogram is used where a return value would normally be used, but the type may only be an unsigned integer in the range of zero to 255. To use this value returned from the child process, symbolic names (constants) are defined for each of the return values used in both Ada and C++ to indicate what type of success or failure has occurred. For instance, the value of zero is defined as *STATUS_SUCCESS* or fully successful completion. For longer messages meant for the user to view (error, warning, or informational messages), the name of a log file is passed as a parameter to the new program. If there is anything in the log file to show the user, an appropriate value is returned. In the case of abnormal termination of the child process, a system error code is automatically returned by *wait*.

Some code needed for results in the GUI had already been written in C++, and by its nature could not easily be segmented to run as another program. Through experience it was found that the C++ Standard Template Library (STL) relied on the *initializers* being called, so they could not reliably be used in any of these routines. However, while resolving this issue, it was also discovered that even though the C++ *initializers* are not run, global and static memory for simple types (ints, chars, pointers, arrays, structs, etc.) was initialized as expected, but the memory occupied by global and static objects (instantiations of classes) is always cleared, initialized to zeroes. There was a need to program defensively, but the clearing of the memory could be used advantageously. If a

Figure 2: *Exception Handling Example for Ada 83*

```
--Top of file (For Ada 83 compiler)
With Current_Exception;
…
-- Exception block
exception
  when others =>
    error_message("Exception " &
        Current_Exception.Exception_Name &
        " propagated out of Export_Overlay");
end Export_Overlay;
```

Figure 3: *Exception Handling Example for Ada 95*

```
--Top of file (For Ada 95 compiler)
With Ada.Exceptions;
…
-- Exception block
exception
  when Event: others =>
    error_message("Exception " &
        Ada.Exceptions.Exception_Name(Event) &
        " propagated out of Export_Overlay");
end Export_Overlay;
```

key variable was still zero, then the object had not been initialized at compile time so it had to be initialized at run time. This may not be true of other compilers, but it is of both of the Ada compilers used for OIET.

There are additional problems when interfacing between Ada and C++. Both languages allow subprogram overloading[4] (including operators), but must work with linkers that require unique symbol names when assembling the program from the separately compiled source files. In order to do this, each compiler makes unique symbol names by *name mangling* (changing the name) but in different ways. For instance, two Ada functions *function calc(it: integer) return integer;* and *function calc(it:float) return float;* in the package *test* become the linker symbols *_A_calc.3S10.test* and *_A_calc.4S10.test* respectively. The equivalent functions declared in C++ become the linker symbols *_Z4calci* and *_Z4calcf* respectively. These exact linker symbols will be different depending on the compilers used.

C does not allow overloading so the function names are not changed to accommodate this feature. Both Ada and C++ provide for linking with code written in the C language. The simplest way to resolve the name mangling issue is to indicate to each compiler that it will be interfacing with C even when there will be no intermediate C function. For C++, the function will also have to be a global function or a static class method.

Different Ada Compilers, One Code Base

Between the Ada 83 and 95 compilers, there are differences in how some language constructs are specified. In the Ada 83 standard, there were suggestions on how to use *pragma* statements to export Ada symbols for use by other languages and how to import symbols from other languages for use by Ada. The Ada 95 standard specified how these interface *pragma*s are to be. But the standard way is different than how our Ada 83 compiler implemented them.

Also, when catching an *OTHERS* exception, Ada 83 provides no standard way to find out what specific exception has been caught. The compiler that we used has a set of functions that will retrieve the exception name and the procedure in which the exception occurred so that we can notify the user (through the GUI) of the problem (see Figure 2). Ada 95 provides a standard way to do this, one different from the proprietary

```
#ifdef C2_2d
With Current_Exception;
#define EXCEPTION_EVENT(x)  x
#define EXCEPTION_NAME  Current_Exception.Exception_Name
#else
#ifdef LCU
With Ada.Exceptions;
#define EXCEPTION_EVENT(x)  Event: x
#define EXCEPTION_NAME  Ada.Exceptions.Exception_Name(Event)
#else
#error "Must define Core version number"
#endif
#endif
```

Figure 4: *Significant Portion of "exceptions.a"*

```
--Top of file (Before preprocessing)
#include exceptions.a
…
-- Exception block
exception
  when EXCEPTION_EVENT(others) =>
    error_message("Exception " &
        EXCEPTION_NAME &
        " propagated out of Export_Overlay");
end Export_Overlay;
```

Figure 5: *Exception Handling Example for Preprocessor*

method our Ada 83 compiler used to implement this feature (see Figure 3).

Both the exception differences and the interface *pragma* differences are issues with parts of the language so they could not be fixed by using an intermediate package. In C or C++, these types of differences would usually be handled by using preprocessor directives to perform conditional compilation and/or macros. The Ada 83 compiler that we are using has a proprietary preprocessor similar to the C preprocessor but using Ada-like syntax. The Ada 95 compiler provides nothing like a preprocessor. Several other solutions were considered, but it was decided that using a preprocessor would be the simplest to implement. To illustrate what was done, the exception example will be used because it is slightly more complex.

The file preprocessor included with a C compiler is simple in theory, substituting text and macros where the preprocessing symbol appears. But when the GNU C compiler was used with the *preprocess only* directive, it objected to the Ada code surrounding the preprocessor code. The result was the same for the GNU C++ compiler.

Brief consideration was given to writing a preprocessor to meet our needs, but to do it right seemed like a two to three week task. In the hope that someone else had encountered a similar problem and already crafted a solution, a search was undertaken at the two largest open-source repositories on the internet: <www.sourceforge.net> and <www.freshmeat.net>. After exploring a few of the resultant programs, *filepp* was found on the later site. It turned out that it does exactly what is needed but is also highly configurable (in case it does not do what is needed)[5,6].

To use the preprocessor to do what needs to be done, a file *exceptions.a* was created that contains the substitutions that need to be done. This file is written such that if no version is specified, the preprocessor will display an error (see Figure 4). To use this file, an *#include exceptions.a* is placed at the beginning of the file where the other *with* directives occur. The exception-handling portion of the code is then rewritten to use the macros (see Figure 5). For instance, *EXCEPTION_EVENT(x)* is a macro expecting text between the parenthesis when used. This text will then be placed where x appears in the text at the end of the line (compare Figures 2 through 5).

If this were used on the file *test.a* to produce the preprocessed version of the file (*test_p.a*) for the Ada 95 compiler (that we call Life Cycle Upgrade [LCU]), the command *filepp -DLCU -o test_p.a test.a* would run. Again, this would be placed in the make file so that the preprocessing is automatically performed

on *test.a* when the program is built. The Ada compiler receives the file *test_p.a*.

Conclusion

The reality of legacy development environments and systems is that not all programming problems can or should be accomplished in the same language, development environment or even the latest versions of these tools. The real world just is not that simple. And sometimes we have to choose the best tool for the job from those available. Interfacing with one or more other languages also requires knowledge of data representations and how the languages send and receive the data. Solutions can be found that will allow maintaining the same code base on two (or more) disparate operating environments without too much maintenance overhead.◆

Notes

1. Both the Ada 83 and Ada 95 standards allow for automatic garbage collection but do not require that the memory be reclaimed until after the type goes out of scope (see <www.adaic.org/docs/craft/html/ch11.htm> and <www.adaic.org/docs/craft/html/ch11.htm>). Since VString is declared in a package, it will not go out of scope until the program exits.
2. C and C++ use the ASCII null character as a sentinel to mark the end of strings. C++ has also defined a more Ada-like string type as part of its standard.
3. *Make* is a program commonly used to build programs from the component source files. The *make file* or *makefile* describes to make the order and how to process each file to make the final result.
4. *Overloading* is having two (or more) subprograms with exactly the same name but different parameter types. The compiler determines which subprogram to call by the types of the parameters passed.
5. See the documentation at <www.cabaret.demon.co.uk/filepp/>.
6. The substitutions done by *filepp* are case-sensitive. In the case of these files, the C convention of making preprocessor symbols all uppercase is used.

About the Author

Val C. Kartchner is the lead programmer of the Overlay Import/Export Tool installable software module for the Air Force Mission Planning System. He has more than 15 years experience in software development, design, and maintenance at Hill AFB, the last six years of which is directly for the Department of Defense. This experience in several different programming languages and development environment has granted the experience necessary to solve issues such as those detailed in this article. Kartchner has a Bachelor of Science in computer science from Weber State University.

309th SMXG/MXDDA
6137 Wardleigh RD
Hill AFB, UT 84056-5843
Phone: (801) 775-2777
Fax: (801) 775-2772
E-mail: val.kartchner@hill.af.mil

Web Sites

ACM SIGAda

www.acm.org/sigada

Here you will find information on the special interest group (SIG) Ada organization and pointers to current information and resources for the Ada programming language. It is a resource for the software community's ongoing understanding of the scientific, technical, and organizational aspects of the Ada language's use, standardization, environments and implementations. This is ACM SIGAda's latest effort to help expand accessibility to Ada information. They want to make this the one stop for information on both SIGAda's current activities and on the Ada language and community at large.

The Ada Information Clearinghouse

http://adaic.org/

The Web site provides articles on Ada applications, databases of available compilers, current job offerings, and more. The Ada Information Clearinghouse is managed by the Ada Resource Association, a group of software tool vendors who support the use of Ada for excellence in software engineering.

Ada Home

www.adahome.com

Since March 1994, this server provides a home to users and potential users of Ada, a modern programming language designed to support sound software engineering principles and practices. The Ada Home Floors and Rooms contain many unique tools and resources to help expand knowledge and increase productivity.

Ada World

www.adaworld.com

Ada World has been created essentially to bring the Ada programming language a central place where Ada developers and curious programmers can learn about Ada, see what is happening as far as Ada development projects go, and give a good idea of what can be done with Ada. To reach this goal, Ada World serves as a place where Ada developers can talk about Ada as well as work on development projects.

Ada Power

www.adapower.com

Ada possesses the ultimate in flexibility (oo and non-oo), real standardization, and validation, true cross-platform programming, incredible compile time error checking, readable code, and support of all levels of software engineering. As a way of contributing back to the Ada community and to help advocate this powerful language, AdaPower.com was formed, and includes on its Web site examples of Ada source code that illustrate various features of the language and programming techniques, various interfaces to popular operating systems (thick and thin level bindings), and examples of Ada source code that illustrate various algorithms; a collection of packages for reuse in Ada programs; and articles on implementing software in Ada.